Lowering the learning curve for declarative programming: a Python API for the IDP system

Joost Vennekens

KU Leuven, Dept. Computerscience @ Technology Campus De Nayer
J.-P. De Nayerlaan 5, 2860 Sint-Katelijne-Waver, Belgium

Abstract. Programmers may be hesitant to use declarative systems, because of the associated learning curve. In this paper, we present an API that integrates the IDP Knowledge Base system into the Python programming language. IDP is a state-of-the-art logical system, which uses SAT, SMT, Logic Programming and Answer Set Programming technology. Python is currently one of the most widely used (teaching) languages for programming. The first goal of our API is to allow a Python programmer to use the declarative power of IDP, without needing to learn any new syntax or semantics. The second goal is allow IDP to be added to/removed from an existing code base with minimal changes.

1 INTRODUCTION

In many contexts where software is currently developed, programmer time is more valuable than computer time. In other words, it is more important to quickly and reliably produce working code, than to optimize the code's runtime. In addition to standard software engineering practices, declarative methods could play an important role in reducing development time. Indeed, a declarative specification of the end result that should be produced contains, in a sense, the minimum of information that must somehow be made available to the computer in order for it to be able to produce the desired output. Moreover, in addition to writing a program from scratch, maintaining and updating a program are typically also quite time consuming tasks. Here too, declarative methods may offer significant advantages, due to their inherent modularity.

In light of these observations, we may expect that recent years would have shown a significant increase in the use of declarative methods throughout industrial software engineering practice. However, evidence to this effect seems to lacking. There may be many reasons for this. Perhaps companies *are* frequently using declarative methods, but prefer to keep this information hidden. Or, perhaps declarative methods are not being widely used because the majority of software engineers work on simple "CRUD" (create-read-update-delete) applications, for which these methods are overkill.

In this paper, we posit the hypothesis that there does exist a larger potential for declarative methods in software engineering than is currently being exploited, and that this potential is not being realised because of the following two contributing factors:

- Many programmers are not familiar with state-of-the-art declarative systems. While their education may have contained, say, an introductory course on Prolog programming, there would still be a substantial learning effort required before they could,

e.g., solve real-world problems by means of, e.g., a modern Answer Set Programming (ASP) [10] or SMT [4] solver.
– Programming typically takes place in a larger context, where there are coworkers to be collaborated with, external systems to be interfaced with, users that need visualisations, etc. If a declarative solution cannot be easily integrated with existing code, it might be practically impossible to adopt it.

In this paper, we investigate how we might integrate a state-of-the-art declarative system within a well-known and widely-used host language, such that:

– There is essentially no learning curve for programmers who already know the host language.
– The interface between the declarative system and the imperative host language uses the existing syntax and semantics of host-language objects, so that the declarative system may easily be replaced by a piece of host-language code, should this ever prove necessary.

In this way, the resulting API fixes the two problems mentioned above and may therefore contribute to a wider adoption of declarative methods in industry. This may prove especially useful for fast prototyping, where declarative systems may offer a substantial benefit. In addition, our API may also offer a convenient way for teachers to introduce students to declarative methods.

In Section 2, we first discuss why we have chosen our particular combination of declarative system and host language. Section 3 then examines to what extent the host language offers objects and expressions that correspond to the inputs needed by the declarative system. Based on this analysis, Section 4 then presents the API that we have developed. Section 5 briefly discusses some notes on its current implementation. In 6, we present several use cases that demonstrate how the API may be used. Section 7 discusses related work. We conclude in Section 8.

Part of this work has been presented to a Logic Programming audience at the *International Workshop on User-Oriented Logic Programming (2015)* of the *International Conference on Logic Programming (ICLP) 2015*. The present paper extends this work by a more thorough discussion of the approach, a better comparison to related work and more illustrative examples.

2 CHOICE OF LANGUAGES

There exists an important distinction between declarative *programming* languages (such as Prolog) and declarative *specification* languages (such as Answer Set Programming). The first kind of languages has an associated operational semantics, which allows algorithms to be specified in the language. By contrast, the second kind lack such an operational semantics, which makes these languages "purely" declarative: a user can specify knowledge about a problem domain, but he cannot express computations.

Both kinds of languages have their own advantages. Because this paper considers the integration of a declarative system with an imperative host language, it makes the most sense to use a declarative specification language. In this way, we obtain a clean

separation between imperative and declarative aspects, which allows us to benefit to the fullest from the advantages of the declarative approach.

There exist many declarative specification languages: ASP [10], SAT/SMT [4], ProB [2], etc. In order to achieve our stated goals, we choose a language that is based as much as possible on classical first-order logic. This will allow us to use the boolean connectives and quantifiers that are part of the host language, without changing their semantics. In this paper, we have selected to use the IDP system [3]. The input language of this system, denoted as FO($\cdot$), is a conservative extension of classical first-order logic, with features such as aggregates, a type system, arithmetic and inductive definitions. As we will show in the next section, this input language can be seamlessly integrated into our chosen host language.

As our host language, we choose Python. On the one hand, this is a suitable choice because we need a somewhat flexible host language in order to be able to achieve an elegant integration. On the other hand, we also want to use a language that is widespread and well-known. Python is reported to be the most popular teaching language for introductory computer science courses[1] and the third most popular programming language overall[2]. We have chosen to use version 2.7 (instead of 3.x), because most of the teaching material currently in use still seems to make use of this version.

In the next section, we explore how the FO($\cdot$) input language of the IDP system can be represented in Python.

3 FINITE FIRST-ORDER LOGIC IN PYTHON

A vocabulary Σ of first-order logic (FO) consists of a set of function symbols and a set of predicate symbols. The FO($\cdot$) language uses a typed variant of FO, which allows formulas to be written in a more compact way, while also helping to avoid errors. In this variant, a number of the unary predicate symbols are designated as *types*, and each other predicate symbol P with arity n (as well as each function symbol F with arity n) is given a typing $P(T_1, \ldots, T_n)$ (respectively, $F(T_1, \ldots, T_n) : T_0$), which defines the types of its arguments (and its range, in case of a function symbol).

A finite structure S for a vocabulary Σ consists of a finite domain D and an assignment to each symbol $\sigma \in \Sigma$ of an interpretation σ^S. If P is a predicate symbol of arity n, then P^S is an n-ary relation on D. The interpretations T_i^S of all the types $T_i \in \Sigma$ must form a partition of the domain D of S. In addition, the interpretation P^S of each predicate $P(T_1, \ldots, T_n)$ must be well-typed, i.e., $P^S \subseteq T_1^S \times \cdots \times T_n^S$. Similarly, the interpretation F^S of a function symbol $F(T_1, \ldots, T_n) : T_0$ is a function $F^S : T_1^S \times \cdots \times T_n^S \to T_0^S$.

In Python, we can use sets of tuples to represent the interpretation of a predicate symbols and dictionaries to represent the interpretation of a function symbol. To illustrate, we consider the example of representing and solving a sudoku puzzle. The puzzle

[1] `http://cacm.acm.org/blogs/blog-cacm/176450-python-is-now-the-most-popular-introductory-teaching-language-at-top-us-universities/fulltext`

[2] `http://spectrum.ieee.org/computing/software/the-2016-top-programming-languages`

grid consists of 81 cells, which are subdivided into 9 rows, 9 columns and 9 smaller 3×3 squares. The layout of this grid can be represented by a type $Cell$ and binary predicates $SameRow$, $SameCol$ and $SameSqu$, each with typing $(Cell, Cell)$. In Python:

```
Cell = range(81)
SameRow = [ (i, j) for i in Cell for j in Cell
                   if i % 9 == j % 9 ]
SameCol = [ (i, j) for i in Cell for j in Cell
                   if i / 9 == j / 9 ]
SameSqu = [(i, j) for i in Cell for j in Cell
          if i/3 == j/3 and (i%9)/3 == (j%9)/3]
```

To represent the numbers that are given in the sudoku grid, we use a (total) function $Given(Cell) : Number$, with the type $Number$ ranging over 0 to 9, where 0 is assigned to the empty cells.

```
Number = range(9)
Given = { 0: 8, 1: 5, 3: 0, 4: 0, 5: 0 ... }
```

The task of solving a sudoku puzzle is that of finding a solution, represented by e.g. a function $Sol(Cell) : Number$, that satisfies all the necessary constraints.

```
Sol = { 0: 8, 1: 5, 3: 2, 4: 3, 5: 6 ... }
```

The constraints that must be satisfied by Sol can be expressed by first-order formulas over the vocabulary Σ. For instance:

$$\forall x[Cell], y[Cell]\ \big(SameRow(x,y) \vee SameCol(x,y) \vee SameSqu(x,y) \Rightarrow Sol(x) \neq Sol(y) \vee x = y\big). \quad (1)$$

Here, the notation $x[Type]$ is used to indicate the type of the quantified variables. Obviously, this information can also be derived automatically from the typing of the predicates.

The following table shows how we can translate the logical connectives into Python expressions:

Formula ϕ	Python expression ϕ^{py}
$P(t_1, \ldots, t_n)$	`(t1,...,tn) in P`
$\neg\phi$	`not` ϕ^{py}
$\phi \vee \psi$	ϕ^{py} `or` ψ^{py}
$\phi \wedge \psi$	ϕ^{py} `and` ψ^{py}
$\forall x[Type] : \psi \Rightarrow \phi$	`all(`ϕ^{py} `for x in Type if` ψ^{py}`)`
$\exists x[Type] : \psi \wedge \phi$	`any(`ϕ^{py} `for x in Type if` ψ^{py}`)`

For instance, formula (1) corresponds to:

```
all(Sol[x] != Sol[y] or x == y for x in Cell for y in Cell
    if (x,y) in SameRow or (x,y) in SameCol or (x,y) in SameSqu)
```

In addition to satisfying this property, Sol also has to coincide with $Given$ for all cells where the latter function is not 0:

```
not any(Sol[x] != Given[x] for x in Cell if Given[x] != 0)
```

Moreover, Sol should not leave any cells empty (i.e., 0 should not occur in its range):

```
all(Sol[x] != 0 for x in Cell)
```

If the above three Python expressions all evaluate to `true`, then Sol is a correct solution to the sudoku instance described by $Given$.

The input language FO(·) of the IDP system extends classical first-order logic with a number of additional features. Most of the *aggregates* it supports are also part of Python, e.g., `min`, `max` and `sum`. Another interesting feature of FO(·) are its *inductive definitions*. An example is the definition of the transitive closure T of a graph G:

$$\left\{ \begin{array}{ll} \forall x[Node], y[Node] & T(x,y) \leftarrow \exists z\, T(x,z) \wedge T(z,y). \\ \forall x[Node], y[Node] & T(x,y) \leftarrow G(x,y). \end{array} \right\}$$

In FO(·), this definition expresses that T is the least fixpoint of the operator induced by the two rules. A similar construct is not readily available in Python, but we can explicitly compute the least fixpoint, using a λ-expression that corresponds to the disjunction of the two rules of the definition.

```
def lfp(f, x=[]):
    y = f(x)
    return y if y == x else lfp(f,y)

node_pairs = [(x,y) for x in Node for y in Node]

d = lambda T: (lambda (x,y): ((x,y) in G or
               any((x,z) in G and (z,y) in T for z in Node)))

TC = lfp(lambda T: filter(d(T), node_pairs))
```

In addition to such monotone inductive definitions, IDP also allows non-monotone inductive definitions, such as definitions over a well-founded order. The IDP system interprets such non-monotone definitions by, essentially, the well-founded model semantics [17]. As shown in [5], this coincides with how such definitions are interpreted in mathematical texts. Again, also this computation could be done explicitly in Python, using a λ-expression that corresponds to the disjunction of the rules of the definition.

Having examined how we can express various parts of FO(·) in Python, we now present our Python API to the IDP system.

4 PYTHON INTERFACE TO THE IDP SYSTEM

The central concept in the API is that of a *knowledge base* (KB). A new KB can be created as follows.

```
from pyidp.typedIDP import IDP
kb = IDP()
```

A KB consists of a vocabulary, a structure for (part of) this vocabulary and a number of constraints that must be satisfied. For instance, the vocabulary and structure for the sudoku example can be added to the KB as follows:

```
kb.Type("Cell", range(81))
kb.Type("Number", range(10))
kb.Predicate("SameRow(Cell,Cell)", [ (x,y) for ... ] )
kb.Predicate("SameCol(Cell,Cell)", [ (x,y) for ... ] )
kb.Predicate("SameSqu(Cell,Cell)", [ (x,y) for ... ] )
kb.Function("Given(Cell): Number", { 0: 8, ... })
kb.Function("Sol(Cell): Number", { 0: 8, ... })
```

Each of these statements adds a symbol to the vocabulary of the KB and assigns an interpretation to it (using the same Python expressions as in Section 3). The API also allows to first declare the symbol and later use the assignment operator = to assign it a value. Once a symbol σ has been added to a KB `kb`, it can be referred to as `kb.`σ. Predicates implement the *MutableSet* interface, while functions implement *Mapping*. This allows these logical object to behave as Python programmers would expect, e.g.:

```
kb.SameRow.add((0,1))
kb.Given[0] = 7
```

This also allows us to evaluate the boolean Python expressions that correspond to the rules of sudoku:

```
all(kb.Sol[x] != kb.Sol[y] or x == y
    for x in kb.Cell for y in kb.Cell if (x,y) in kb.SameRow
         or (x,y) in kb.SameCol or (x,y) in kb.SameSqu)
```

Instead of immediately evaluating this expression, we can also add it as a *constraint* to the knowledge base:

```
kb.Constraint(
 """all(Sol[x] != Sol[y] or x == y
    for x in Cell for y in Cell if (x,y) in SameRow
         or (x,y) in SameCol or (x,y) in SameSqu)""")
```

Each KB has a boolean property `satisfiable` that can be checked to find out if all constraints that have been added to it are in fact satisfied by the KB's current interpretation of the vocabulary.

```
if kb.satisfiable:
    print "Sudoku is solved."
```

From a logical perspective, inspecting the value of this property triggers IDP's inference task of *model checking*: for the finite structure S that is represented by the KB and for the constraints $\phi_1, \ldots, \phi_n$ that belong to the KB, it is checked whether $S \models \bigwedge_{1 \leq i \leq n} \phi_i$.

In addition to this inference task, IDP also supports *model expansion*: given a structure S_0 for a subvocabulary $\Sigma_0 \subseteq \Sigma$ of the vocabulary of the formulas $\phi_1, \ldots, \phi_n$, compute a structure S for the vocabulary $\Sigma \setminus \Sigma_0$ such that $(S_0 \cup S) \models \phi_1 \wedge \cdots \wedge \phi_n$. This task is known to capture the complexity class NP [13].

Our API supports this inference task in a very simple way: if the programmer declares a vocabulary symbol but does not assign an interpretation to it, then any attempt to inspect the value of this symbol will trigger a call to IDP's model expansion algorithm. This will then automatically fill in the interpretation of this symbol in accordance with the constraints. If multiple different interpretations are possible, one is arbitrarily selected (but, if there are multiple such symbols, then the same model is used to generate the interpretation for all of them, so that the interpretations for different symbols are always mutually consistent). In case of the sudoku example, if we had only declared the symbol Sol, without assigning it a value:

```
kb.Function("Sol(Cell): Number")
```

then the following code would compute and print a solution to the given sudoku instance:

```
for x in kb.Cell:
    print kb.Sol[x]
```

A final feature of our API is that it also supports the *inductive definitions* of FO($\cdot$). The method `Define` may be used to at once declare a symbol and define it by means of a λ-expression. In the context of the sudoku example, we may use this to define the following auxiliary concept:

```
kb.Define("Diff(Cell,Cell)",
    """lambda x,y: (x,y) in SameRow
                or (x,y) in SameCol or (x,y) in SameSqu""")
```

In words, this statement defines that `Diff` is the set of all pairs of cells (x, y) for which the given λ-expression holds. Once this concept has been defined, it can be used to simplify the main sudoku constraint.

```
kb.Constraint("""all(Sol[x] != Sol[y]
              for (x,y) in Diff if x != y)""")
```

Even though the above example does not demonstrate this, in general, these definitions may be inductive. For instance, the definition of transitive closure can be given as:

```
kb.Define("T(Node,Node)",
  """lambda x,y: (x,y) in G
     or any((x,z) in G and (z,y) in T for z in Node)""")
```

Discussion. The above API allows the IDP system to be used from Python without requiring any knowledge that a Python programmer does not already possess: semantical objects (i.e., interpretations of predicate and function symbols) take the form of standard Python sets and dictionaries, while constraints take the form of standard Python

boolean expressions, and definitions make use of standard Python λ-expressions. All of these Python objects and expressions can be used in all of the normal ways and retain their normal semantics. Moreover, even those arguments that are passed as strings (e.g., to the `Constraint` method) are handled by the Python parser within the API, so standard syntactical rules apply and standard messages are generated for syntax errors.

In order to make effective use of this API, a Python programmer therefore only has to know two things:

- The property `satisfiable` of a KB is `true` if and only if all Python expressions that were added as constraints would normally (i.e., under their Python semantics) evaluate to `true`;
- If some symbol of a KB is not assigned a value, then a value will be automatically computed in such a way that all of the constraints would evaluate to `true`.

A small exception to the above discussion is that our API of course also requires the programmer to declare a typed vocabulary. This is something which has no counterpart in the dynamically typed Python language, and which therefore also requires some additional explanation. However, given the simplicity of the type system, this should be trivial to understand for any programmer.

In summary, we may therefore conclude that the API should be immediately usable with minimal learning effort.

5 NOTES ON IMPLEMENTATION

The implementation of our API is available for download from the following URL:

```
https://bitbucket.org/joostv/pyidp
```

Also the examples presented in this paper can be found here.

Interfacing with the IDP system is currently done in a decoupled way: when the API detects that the IDP system needs to be called, it prepares a text file with the appropriate vocabulary, structure and theory; it then calls IDP as an external process and parses its output. The results of this call are cached, so that IDP is not invoked again, as long as the KB does not change. Obviously, a tighter integration, which avoids calling IDP as an external process and communicating through text files, would improve the efficiency of the API. Moreover, a tight integration might be developed which would allow us to keep an instance of the IDP system running, such that only the differences with the previous invokation need to be communicated when a new invokation is needed.

The IDP system offers various options which can be used to speed up certain computations. For instance, it can make use of the XSB Prolog system[3] [14] to handle inductive definitions. The KB objects offered by our API have a method `set_idp_option` that can be used to set such options. For instance, XSB is enabled by:

```
kb.set_idp_option("xsb", "true")
```

[3] `http://xsb.sourceforge.net/`

6 USE CASES

In this section, we examine several ways in which the API can be used. We pay particular attention to the ease with which our API can be integrated into existing Python code and the functionality that it can deliver. Because we specifically aim at reducing the programming effort in situations where efficiency is of secondary importance (such as prototyping), we do not investigate computational complexity. In general, however, solutions using our API will of course be significantly slower than purpose-built algorithms in the host language (but comparable to using IDP as a stand-alone system).

6.1 Solving combinatorial problems

A typical use case for declarative systems is the solving of combinatorial problems. As an example, we consider the problem of solving Hidato puzzles. The goal of such puzzles is to fill in the numbers 1 to n in a grid of n cells, such that each i and $i+1$ are in adjoining cells (horizontally, vertically or diagonally), taking into account the fact that the position of certain numbers is fixed up-front. We have taken a Python solver for such puzzles that is available online[4] and adapted it to our API.

As a vocabulary, we use types to represent the rows (`R`), columns (`C`) and numbers (`Nb`) that need to be entered in the cells. The predicate `Cell(R,C)` describes which combinations of row and column numbers corresponds to cells, while `Given(R,C,Nb)` gives the numbers that are already filled in. We will describe the solution to the puzzle by means of functions `Row` and `Col` that map each number to the row/column in which this number is filled in.

```
hid = IDP()
hid.Type("R", [1])
hid.Type("C", [1])
hid.Type("Nb", [1])
hid.Predicate("Cell(R,C)", [])
hid.Predicate("Given(R,C,Nb)", [])
hid.Function("Row(Nb): R")
hid.Function("Col(Nb): C")
```

As constraints, we first need to express that the solution must coincide with the numbers that are given.

```
hid.Constraint("all(Row(v) == r and Col(v) == c for (r,c,v) in Given)")
```

Next, each cell may only contain one number and numbers may only appear in the cells:

```
hid.Constraint(
  """all(c == d for c in Nb for d in Nb
         if Row(c) == Row(d) and Col(c) == Col(d))""")
hid.Constraint("all(Cell(Row(n),Col(n)) for n in Nb)")
```

[4] http://rosettacode.org/wiki/Solve_a_Hidato_puzzle#Python

Finally, there is the constraint that each number must be in the Moore neighbourhood of its successor.

```
hid.Constraint(
  """all(abs(Row(c) - Row(c+1)) < 2
         and abs(Col(c) - Col(c+1)) < 2
         for c in Nb if c < max(Nb))""")
```

The Python solution from which we start defines three functions: `setup` initialises the data structure representing the puzzle, `solve` computes the solution and `printboard` visualises it. By making a few small changes to `setup` and `printboard`, we can make these functions use the KB `hid` constructed above. We thereby replace the entire `solve` function, as the solution will now be computed by the IDP system as soon as `printboard` tries to visualise it.

```
def setup(s):
    lines = s.splitlines()
    hid.C = range(len(lines[0].split()))
    hid.R = range(len(lines))
    cellcount = 1
    for r, row in enumerate(lines):
        for c, cell in enumerate(row.split()):
            if cell == ".": # not a cell
                continue
            hid.Nb.add(cellcount)
            hid.Cell.add((r,c))
            if cell != "__": # cell not empty
                hid.Given.add((r,c,int(cell)))
            cellcount += 1

def print_board():
    d = {-1: "  ", 0: "__"}
    bmax = max(hid.Nb)
    form = "%" + str(len(str(bmax)) + 1) + "s"
    matrix = [[' ' for i in hid.C]
                           for i in hid.R]
    for c in hid.Nb:
        matrix[hid.Row[c]][hid.Col[c]] = c
    for r in matrix:
        print " ".join(map(lambda x:form%x,r))
```

We can now solve a Hidato puzzle as follows:

```
hi = """\
__ 33 35 __ __  .  .  .
__ __ 24 22 __  .  .  .
__ __ __ 21 __ __  .  .
__ 26 __ 13 40 11  .  .
```

```
27 __ __ __  9 __  1  .
 .  . __ __ 18 __ __  .
 .  .  .  . __  7 __ __
 .  .  .  .  .  .  5 __"""

setup(hi)
print_board()
```

6.2 Working with graphs

The following class `GraphKB` extends the generic IDP Knowledge Base class with some specific functionality for working with undirected graphs. When constructing such a `GraphKB`, the nodes of the graph can be initialised by means of a given set and the edges by means of an adjacency list. The predicate `Edge` is defined as the symmetric closure of the adjacency list. This class also offers a convenience method to define the transitive closure of a given relation.

```
class GraphKB(IDP):

    def __init__(self, nodes=[0], adj_list=[]):
        super(GraphKB, self).__init__()
        self.Type("Node", nodes)
        self.Predicate("Adj(Node,Node)", adj_list)
        self.Define("Edge(Node,Node)",
           "lambda x,y: Adj(x,y) or Adj(y,x)")

    def def_TC(self, original, tc_name):
        formula = """lambda x,y: {0}(x,y) or
         any({1}(x,z) and {1}(z,y)
             for z in Node)""".format(original, tc_name)
        self.Define(tc_name+"(Node, Node)", formula)
```

We can now check if a given adjacency list describes a fully connected graph:

```
conn = GraphKB(nodes, adj)
conn.def_TC("Edge", "Path")
conn.Constraint("all(Path(x,y) for x in Node for y in Node)")
if conn.satisfiable:
    print "Graph is fully connected"
```

We can use a similar KB to count the number of connected components in the graph. We do this by selecting a single representative from each component (its "`Root`") and then counting the number of these representatives.

```
cc = GraphKB(nodes, adj)
cc.def_TC("Edge", "Path")
cc.Predicate("Root(Node)")
```

```
cc.Constraint("""all(any(Path(r,x) for r in Root)
                 for x in Node if not Root(x))""")
cc.Constraint("""not any(Path(x,y)
                 for x in Root for y in Root if x != y)""")
print "Components: {0}".format(len(comp.Root))
```

In graph theory, an undirected graph is called a *tree* if it is connected and does not contain cycles. When checking for a cycle in an undirected graph, we of course have to exclude the trivial two-node cycles that would result from traversing the same undirected edge in both directions. This in fact makes it easier to use IDP to check that there *is* a cycle, than to check that there *is not* one. The following knowledge base tries to guess the direction in which to traverse each edge in order to produce a cycle. If it is unsatisfiable, there are no cycles.

```
cyclic = GraphKB()
cyclic.Predicate("Traverse(Node,Node)")
cyclic.Constraint("all(Edge(x,y) for (x,y) in Traverse)")
cyclic.Constraint(
 "not any(Traverse(y,x) for (x,y) in Traverse)")
cyclic.def_TC("Traverse", "TravTC")
cyclic.Constraint("any(TravTC(x,x) for x in Node)")
```

We can now combine the two knowledge bases to check whether a given adjacency list indeed describes a tree.

```
def is_tree(adj_list):
    cyclic.Adjacent = adj_list
    conn.Adjacent = adj_list
    return (bool(conn.satisfiable)
            and not bool(cyclic.satisfiable))
```

This example illustrates how additional functionality can be built on top of the KB objects of our API. In addition, the ability to combine the results of calls to different KBs also allows us to implement functionality that would be harder to implement in a single IDP KB.

6.3 Flexible input/output

Bio-informatics applications may need to translate between strings of bases and strings of amino acids. In this translation, a *codon* (i.e., a sequence of three bases) corresponds to a single amino acid, according to a fixed and well-known table. For instance, the following nine bases correspond to the following three amino acids:

$$\underbrace{a\,c\,t}_{\mathrm{T}}\;\underbrace{g\,a\,g}_{\mathrm{E}}\;\underbrace{t\,c\,a}_{\mathrm{S}}$$

The following knowledge base declaratively defines the relation between the two different kinds of sequence. Here, the sequences are represented by mappings of indices to, respectively, bases and amino acids.

```
k = IDP()
k.Type("Base", ['t', 'c', 'a', 'g'])
codons = [(a,b,c) for a in k.Base
          for b in k.Base for c in k.Base]
amino_acids = ('FFLLSSSSYY**CC*WLLLLPPPPHHQQRRRR'+
               'IIIMTTTTNNKKSSRRVVVVAAAADDEEGGGG')
k.Type("AmAcid", set(amino_acids))
k.Type("AIndex")
k.Type("BIndex")

k.Function("Codon(Base, Base, Base): AmAcid",
               dict(zip(codons, amino_acids)))
k.Function("BaseAt(BIndex): Base")
k.Function("AmAcidAt(AIndex): AmAcid")
k.Constraint(
  """all(Codon(BaseAt(3*i), BaseAt(3*i+1), BaseAt(3*i+2))
         == AmAcidAt(i) for i in PIndex)""")
```

The following function translates a regular Python list into a dictionary that maps list indices to values. It will be convenient to construct an interpretation for the `BaseAt` and/or `AmAcidAt` functions.

```
def sequence(list_):
    return dict(enumerate(list))
```

Using the above knowledge base, we can translate bases to amino acids as follows:

```
bases = 'actgagtca'
k.BaseAt = sequence(bases)
k.BIndex = range(len(bases))
k.AIndex = range(len(seq) / 3)
print k.AmAcidAt
```

Because of its purely declarative nature, the same knowledge base can also be used to perform the translation in the other direction.

```
amino_acids = 'TES'
k.AmAcidAt = sequence(amino_acids)
k.PAndex = range(len(seq))
k.BIndex = range(len(seq) * 3)
print k.BaseAt
```

6.4 Self-maintaining data-structures

Whenever the interpretation of one or more symbols in the vocabulary of a knowledge base changes, the API will automatically recompute the interpretation of the other symbols as soon as this is needed. To illustrate, we implement the following simple method of constructing a random fully connected directed acyclic graph:

– While there are still unconnected nodes:
  - Randomly select a pair (x, y) of unconnected nodes
  - Add an edge from x to y

Using the `GraphKB` class of Section 6.2, we can implement this as follows:

```
kb = GraphKB(["a","b","c","d"])
kb.def_TC("Edge", "TC")
kb.Define("Unconnected(Node,Node)",
 """lambda x,y: x != y and not (TC(x,y) or TC(y,x))""")

import random
while len(kb.Unconnected) > 0:
    kb.Edge.add(random.choice(list(kb.Unconnected)))
print kb.Edge
```

Each time the $Edge$ relation is updated in the `while`-loop, the knowledge base is automatically invoked to keep the $Unconnected$ relation up-to-date.

7 RELATED WORK

There is already a long history of work attempting to close the gap between imperative and declarative programming [1]. We briefly compare our approach to some recent work in this area.

Several such approaches exist in the domain of Answer Set Programming. In [9], Python is used a layer on top of the ASP solver Claps, in order invoke this solver in this different ways. In [7], Python and ASP are more tightly coupled: the ASP solver cannot only be invoked from Python, but various pieces of Python code can also be called during the solving process. In contrast to our system, both these approaches expect the Python programs to be written by a knowledgable ASP programmer. The approach of [8] is most similar to ours: it embeds ASP in Java, allowing information contained in standard Java data structures to be completed or checked by means of ASP programs. However, while standard Java data standard structures are used, the ASP programs are still written in their standard syntax, again requiring a knowledgable ASP programmer.

In [16], an approach is presented in which a constraint solver is not added to a single host language, but can be used in the development of a domain-specific language in Racket. Like ours, the motivation behind this work is to allow the power of declarative systems to be more widely used. However, their approach differs, because they count on an intermediary—the designer of the domain-specific language—to hide the complexity of the declarative system, whereas our approach focuses on creating an interface that is natural enough to offer KB functionality directly.

In [11], a constraint solver is integrated into the Scala language. As ours does, their approach reuses the syntax of the host language to interface with the declarative system. A key difference is that, in their approach, the programmer is explicitly manipulating, combining and solving constraints, which makes the constraint solver more present in

the eventual source code. A second difference is of course that Scala currently appears to be less widely known than Python.

In [12], a reasoner for FO extended with transitive closure is integrated into Java. Their KB language is therefore very similar to (but more restricted than) that of IDP. When it comes to the integration in Java, there are two main differences to our approach. First, the declarative knowledge is not written in expressions of the host language, but in a separate language (the Alloy-like JFSL [18]). Second, the integration into Java is done in an object-oriented way: the programmer defines classes in which formulas are added as, among others, class invariants, method pre-/postconditions and frame conditions. In comparison, our Python API seems more lightweight, since it does not require an object-oriented approach. When it comes to computational performance, [12] reports good results, which our implementation is not able to match.

In summary, we believe that our approach fills a niche as an easy-to-learn rapid prototyping API, that, due to Python's current popularity, may speak to a large audience.

8 CONCLUSIONS AND FUTURE WORK

Developing an algorithm to solve a particular computational problem may require a substantial effort. Moreover, it may be time-consuming to adapt such an algorithm to even small changes in the problem specification. The use of a declarative system may therefore provide an interesting alternative, especially in situations were flexibility and development speed are of prime importance (and computational efficiency is not). Typically, this occurs in the prototyping stages of an application.

Programmers may nevertheless be reluctant to use a declarative system for a number of reasons:

- the system may be hard to learn for themselves or for their coworkers;
- generating input for the system in the appropriate format may require a large effort, as may parsing the output of the system and extracting the necessary information from it.

In this paper, we have presented a Python API for the IDP system that avoids these problems. It uses only standard Python objects and expressions, which has two main benefits:

- there is essentially no learning curve: the programmer needs to know nothing about the IDP system or its input syntax in order to make successful use of the API;
- it is easy to incorporate the API into existing Python code, or to replace an existing use of the API by native Python code.

We have presented several use cases of this API, illustrating its use to solve computational problems, to perform various graph computations, to implement flexible input/output behaviour and self-maintaining data structures.

One problem with the current implementation of our API is that there is no support for debugging the declarative specification, which may be especially problematic if the specification contains a bug that makes it inconsistent. In future work, we will address this issue. This will enable us to conduct experiments in which the API is used by programmers who are not familiar with the IDP system.

References

1. Krzysztof R. Apt and Andrea Schaerf. Programming in alma-0, or imperative and declarative programming reconciled. In *FroCos*, 1998.
2. Jens Bendisposto, Joy Clark, Ivaylo Dobrikov, Philipp Körner, Sebastian Krings, Lukas Ladenberger, Michael Leuschel and Daniel Plagge. ProB 2.0 Tutorial In *Proceedings of the 4th Rodin User and Developer Workshop*, TUCS Lecture Notes, 2013.
3. M. Bruynooghe, H. Blockeel, B. Bogaerts, B. De Cat, S. De Pooter, J. Jansen, A. Labarre, J. Ramon, M. Denecker, and S. Verwer. Predicate logic as a modeling language: Modeling and solving some machine learning and data mining problems with IDP3. *Theory and Practice of Logic Programming*, 2014. Accepted.
4. Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient SMT solver. In *Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2008.
5. Marc Denecker and Joost Vennekens. The Well-Founded Semantics Is the Principle of Inductive Definition, Revisited. In *Proceedings of KR*. 2014.
6. Marc Denecker and Eugenia Ternovska. A logic of nonmonotone inductive definitions. *ACM Transactions on Computational Logic*, 9(2), March 2008.
7. T. Eiter, M. Fink, G. Ianni, T. Krennwallner, C. Redl and P. Schüller. A model building framework for Answer Set Programming with external computations. Theory and Practice of Logic Programming, 16(4), 418464, 2016.
8. O. Febbraro, G. Grasso, F. Ricca and N. Leone. JASP: A Framework for Integrating Answer Set Programming with Java. In *KR*, 2012.
9. M. Gebser, R. Kaminski, P. Obermeier and T. Schaub. A Case-Study in Multi-shot ASP Solving. In *Advances in Knowledge Representation, Logic Programming, and Abstract Argumentation* (pp. 17–32). Springer, 2015.
10. Martin Gebser, Benjamin Kaufmann, André Neumann and Torsten Schaub. Clasp: A conflict-driven answer set solver. In *Logic Programming and Nonmonotonic Reasoning* (pp. 260-265). Springer Berlin Heidelberg, 2012.
11. A. Köksal, V. Kuncak, and P. Suter. Constraints as control. In *POPL'12*, 2012.
12. A. Milicevic, D. Rayside, K. Yessenov, and D. Jackson. Unifying execution of imperative and declarative code. In *Proc. 33rd Int'l Conference on Software Engineering (ICSE)*, 2011.
13. David G. Mitchell and Eugenia Ternovska. A framework for representing and solving NP search problems. In *AAAI*, pages 430–435, 2005.
14. T. Swift and D. S. Warren. XSB: Extending Prolog with tabled logic programming. *Theory and Practice of Logic Programming* 12(1-2):157-187, 2012.
15. Shahab Tasharrofi and Eugenia Ternovska. A semantic account for modularity in multi-language modelling of search problems. In *Proc. FroCos*, 2011.
16. E. Torlak and R. Bodik. Growing solver aided languages with ROSETTA. In *Proc. ACM international symposium on New ideas, new paradigms, and reflections on programming & software*, 2013.
17. A. Van Gelder, K.A. Ross, and J.S. Schlipf. The well-founded semantics for general logic programs. *Journal of the ACM*, 38(3):620–650, 1991.
18. K. Yessenov. A lightweight specification language for bounded program verification. Master's thesis, MIT, 2009.